Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6-8 по курсу**

**«Операционные системы»**

Студент: Ткаченко Егор Юрьевич

Группа: М8О-207Б-21

Вариант: 10

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2022

**Содержание**

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Демонстрация работы программы
7. Выводы

**Репозиторий**

[https://github.com/Tnirpps/OS\_lab](https://vk.com/away.php?to=https%3A%2F%2Fgithub.com%2FTnirpps%2FOS_lab&cc_key=)

**Постановка задачи**

**Цель работы**

Целью является приобретение практических навыков в:  
 Управлении серверами сообщений (№6)  
 Применение отложенных вычислений (№7)  
 Интеграция программных систем друг с другом (№8)

**Задание**

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать два вида узлов: «управляющий» и «вычислительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить при помощи технологии очередей сообщений. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом. При убийстве («kill -9») любого вычислительного узла система должна пытаться максимально сохранять свою работоспособность, а именно все дочерние узлы убитого узла могут стать недоступными, но родительские узлы должны сохранить свою работоспособность.

Управляющий узел отвечает за ввод команд от пользователя и отправку этих

команд на вычислительные узлы. Список основных поддерживаемых команд:

• Создание нового вычислительного узла;

• Удаление существующего вычислительного узла;

• Исполнение команды на вычислительном узле;

• Проверка доступности вычислительного узла.

**Задание варианта**

Вариант 10.

Все вычислительные узлы находятся в списке. Есть только один управляющий узел.

Исполнение команды — поиск подстроки в строке.

Команда проверки — проверка доступности всех узлов.

**Общие сведения о программе**

Программа распределительного узла компилируется из файла main.c, программа вычислительного узла компилируется из файла node.c. В программе используется библиотека для работы с сервером сообщений ZeroMQ. В программе используются следующие системные вызовы:

fork –– создает новый процесс, который является копией родительского процесса, за исключением разных process ID и parent process ID. В случае успеха fork() возвращает 0 для ребенка, число больше 0 для родителя – child ID, в случае ошибки возвращает -1.

execl — используется для выполнения другой программы. Эта другая программа, называемая процессом-потомком (child process), загружается поверх программы, содержащей вызов exec. Имя файла, содержащего процесс-потомок, задано с помощью первого аргумента. Какие-либо аргументы, передаваемые процессу-потомку, задаются либо с помощью параметров от arg0 до argN, либо с помощью массива arg[].

Также были использованы следующие вызовы из библиотеки ZMQ:

zmq\_ctx\_new –– создает новый контекст ZMQ.

zmq\_connect — создает входящее соединение на сокет.

zmq\_disconnect — отсоединяет сокет от заданного endpoint’a.

zmq\_socket — создает ZMQ сокет.

zmq\_close — закрывает ZMQ сокет.

zmq\_ctx\_destroy –– уничтожает контекст ZMQ.

**Общий метод и алгоритм решения**

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы с ZMQ.

2. Проработать принцип общения между клиентскими узлами и между первым клиентом и сервером и алгоритм выполнения команд клиентами.

3. Реализовать необходимые функции-обертки над вызовами функций библиотеки ZMQ.

4. Написать программу сервера и клиента

**Исходный код**

================================== main.c ==================================

#include <stdio.h>

#include "../headers/msgQ.h"

int\* NODES;

int split\_copy(const char\* text, char\* dest, int index) {

int i = 0;

for (int j = 0; j < index; ++j) {

// skip word in text

while (text[i] != ' ' && text[i] != '\0' && text[i] != '\n') {

++i;

}

if (text[i] == ' ') {

++i;

}

}

int k = i;

while (text[i] != ' ' && text[i] != '\0' && text[i] != '\n') {

dest[i - k] = text[i];

++i;

}

dest[i - k] = '\0';

return i - k;

}

int \_\_str\_to\_int(const char\* str) {

int res = 0;

int protect = 0;

for (int i = 0; str[i] > 0; ++i) {

if (str[i] >= '0' && str[i] <= '9') {

if (protect++ > 4) return -100000;

res = res \* 10 + str[i] - '0';

}

}

return res;

}

int node\_exist(int value) {

if (NODES == NULL) return 0;

int i = 0;

while (NODES[i] != TERMINATOR) {

if (NODES[i] == value) return 1;

++i;

}

return 0;

}

void stop\_all\_children(void\* requester, char\* addr, message token) {

for (int i = 0; NODES[i] != TERMINATOR; ++i) {

token.cmd = delete;

if (ping\_process(NODES[i])) {

reconnect\_zmq\_socket(requester, NODES[i], addr);

send\_msg\_wait(requester, &token);

receive\_msg\_wait(requester, &token);

}

}

}

int main (int argc, char const \*argv[]) {

void \*context = zmq\_ctx\_new();

void \*requester = create\_zmq\_socket(context, ZMQ\_REQ);

NODES = create\_vector();

if (NODES == NULL) return 1;

char query\_line [MN];

char query\_word[MN];

char query\_int[MN];

char addr[MN] = SERVER\_SOCKET\_PATTERN;

int last\_created = -1;

printf("ME: %d\n", getpid());

message token = {create, 0, "", ""};

while (fgets(query\_line, MN, stdin) != NULL) {

split\_copy(query\_line, query\_word, 0);

if (strcmp(query\_word, "create") == 0) {

if (split\_copy(query\_line, query\_int, 1) == 0) {

printf("\tbad id, try another one\n");

continue;

}

int id\_process = \_\_str\_to\_int(query\_int) + MIN\_ADDR;

if (node\_exist(id\_process)) {

printf("\tthis node was created earlier\n");

continue;

}

char flag\_creation\_str[MN];

int flag\_creation = 0;

int l = split\_copy(query\_line, flag\_creation\_str, 2);

if (l == 2) {

if (flag\_creation\_str[0] == '-' && flag\_creation\_str[1] == '1') {

flag\_creation = 1;

}

}

if (length(NODES) == 0 || flag\_creation) {

if (id\_process > MIN\_ADDR) {

printf("\tI'm creating %d\n", id\_process);

NODES = push\_back(NODES, id\_process);

if (NODES == NULL) {

printf("\tSm-th wrong with dynamic array\n");

return 1;

}

reconnect\_zmq\_socket(requester, id\_process, addr);

} else {

printf("\tSm-th wrong with number\n");

continue;

}

memset(query\_int, 0, MN);

sprintf(query\_int, "%d", id\_process);

char \*Child\_argv[] = {"node", query\_int, NULL};

int pid = fork();

if (pid == -1) {

return 1;

}

if (pid == 0) {

execv("node", Child\_argv);

return 0;

}

last\_created = id\_process;

} else {

if (id\_process < MIN\_ADDR) {

printf("\tbad id\n");

continue;

}

if (ping\_process(last\_created) == false) {

printf("\tCannot create Node from [%d]\n", last\_created - MIN\_ADDR);

continue;

}

token.cmd = create;

token.value = id\_process;

create\_addr(addr, last\_created);

printf("\tsend to %s\n", addr);

send\_msg\_wait(requester, &token);

receive\_msg\_wait(requester, &token);

if (token.cmd == success) {

NODES = push\_back(NODES, id\_process);

last\_created = id\_process;

if (NODES == NULL) {

printf("\tSm-th wrong with dynamic array\n");

return 1;

}

} else {

printf("\t%d was successfully created\n", id\_process - MIN\_ADDR);

}

}

} else if (strcmp(query\_word, "exec") == 0) {

split\_copy(query\_line, query\_int, 1);

split\_copy(query\_line, query\_int, 1);

int id\_process = \_\_str\_to\_int(query\_int) + MIN\_ADDR;

if (node\_exist(id\_process) && ping\_process(id\_process)) {

fgets(token.str, MAX\_LEN, stdin);

fgets(token.sub, MAX\_LEN, stdin);

for (int i = MAX\_LEN - 1; i >= 0; --i) {

if (token.str[i] == '\n') {

token.str[i] = '\0';

break;

}

}

token.cmd = exec;

reconnect\_zmq\_socket(requester, id\_process, addr);

send\_msg\_wait(requester, &token);

receive\_msg\_wait(requester, &token);

if (token.cmd == success) {

printf("\tall matches: ");

for (int i = 0; i < MAX\_LEN; ++i) {

if (token.res[i] == -1) {

if (i == 0) printf("NO");

break;

}

printf("%d ", token.res[i]);

}

printf("\n");

} else {

printf("\tcannot exec %d\n", id\_process - MIN\_ADDR);

}

} else {

printf("\t[%d] Node hasn't connection\n", id\_process - MIN\_ADDR);

}

} else if (strcmp(query\_word, "remove") == 0) {

split\_copy(query\_line, query\_int, 1);

int id\_process = \_\_str\_to\_int(query\_int) + MIN\_ADDR;

if (node\_exist(id\_process) && ping\_process(id\_process)) {

reconnect\_zmq\_socket(requester, id\_process, addr);

token.cmd = delete;

send\_msg\_wait(requester, &token);

receive\_msg\_wait(requester, &token);

if (token.cmd != success) {

printf("\tcannot remove %d\n", id\_process - MIN\_ADDR);

}

} else {

printf("\t[%d] Node hasn't connection\n", id\_process - MIN\_ADDR);

}

} else if (strcmp(query\_word, "pingall") == 0) {

printf("\t");

for (int i = 0; NODES[i] != TERMINATOR; ++i) {

if (ping\_process(NODES[i]) == true) {

printf("[%d is OKE] ", NODES[i] - MIN\_ADDR);

} else {

printf("[%d is BAD] ", NODES[i] - MIN\_ADDR);

}

}

printf("\n");

} else {

printf("\tBad command. Please, try again\n");

}

memset(query\_line, 0, MN);

memset(query\_word, 0, MN);

memset(query\_int, 0, MN);

}

stop\_all\_children(requester, addr, token);

close\_zmq\_socket(requester);

destroy\_zmq\_context(context);

destroy(NODES);

return 0;

}

============================== client.c ===================================

#include <stdio.h>

#include <string.h>

#include "../headers/msgQ.h"

int\* z\_function(const char\* s) {

int n = (int) strlen(s);

int\* z = malloc(n \* sizeof(int));

if (z == NULL) {

return NULL;

}

int l = 0, r = 0;

z[0] = 0;

for (int i = 1; i < n; ++i) {

if (i <= r) {

// here is z[i] = min(z[i - l], r - i) :

z[i] = (z[i - l] < r - i) ? z[i - l] : r - i;

}

while (i + z[i] < n && s[i + z[i]] == s[z[i]]) {

++z[i];

}

if (i + z[i] > r) {

l = i;

r = i + z[i];

}

}

return z;

}

void find\_substr(message\* token) {

/\*

z\_function(<pattern> + '#' + <text>)

here we use '\n' as '#'

\*/

int n = (int) strlen(token->str);

int m = (int) strlen(token->sub);

char string [n + m + 1];

memcpy(string, token->sub, m);

memcpy(string + m, token->str, n);

string[n + m] = '\0';

int\* z = z\_function(string);

clear\_token(token);

if (z == NULL) {

return;

}

int k = 0;

for (int i = 0; i < n; i++) {

if (z[m + i] == m - 1) {

// one indexing

token->res[k++] = i + 1;

}

}

token->cmd = success;

free(z);

}

int main(int argc, char const \*argv[]) {

if (argc < 2) {

printf("argv error\n");

return 1;

}

void \*context = create\_zmq\_context();

void \*responder = create\_zmq\_socket(context, ZMQ\_REP);

char adr[30] = TCP\_SOCKET\_PATTERN;

strcat(adr, argv[1]);

printf("\t[%d] has been created\n", getpid());

bind\_zmq\_socket(responder, adr);

while (1) {

message token;

receive\_msg\_wait(responder, &token);

int id\_process;

char query\_int[30];

switch (token.cmd) {

case delete:

token.cmd = success;

printf("\t[%d] has been destroyed\n", getpid());

send\_msg\_wait(responder, &token);

close\_zmq\_socket(responder);

destroy\_zmq\_context(context);

return 0;

case create:

id\_process = token.value;

memset(query\_int, 0, 30);

sprintf(query\_int, "%d", id\_process);

char \*Child\_argv[] = {"node", query\_int, NULL};

int pid = fork();

if (pid == -1) {

return 1;

}

if (pid == 0) {

execv("node", Child\_argv);

return 0;

} else {

token.cmd = success;

send\_msg\_wait(responder, &token);

}

break;

case exec:

find\_substr(&token);

send\_msg\_wait(responder, &token);

break;

default:

// equals "delete brunch"

token.cmd = success;

send\_msg\_wait(responder, &token);

close\_zmq\_socket(responder);

destroy\_zmq\_context(context);

return 0;

}

}

}

============================== msgQ.c ================================

#include "../headers/msgQ.h"

void create\_addr(char\* addr, int id) {

char str[MN];

memset(str, 0, MN);

sprintf(str, "%d", id);

memset(addr, 0, MN);

memcpy(addr, SERVER\_SOCKET\_PATTERN, sizeof(SERVER\_SOCKET\_PATTERN));

strcat(addr, str);

}

void clear\_token(message\* msg) {

msg->cmd = delete;

msg->value = 0;

memset(msg->str, 0, MAX\_LEN);

memset(msg->sub, 0, MAX\_LEN);

for (int i = 0; i < MAX\_LEN; ++i) {

msg->res[i] = -1;

}

}

void\* create\_zmq\_context() {

void\* context = zmq\_ctx\_new();

if (context == NULL) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_ctx\_new ");

exit(ERR\_ZMQ\_CTX);

}

return context;

}

void bind\_zmq\_socket(void\* socket, char\* endpoint) {

if (zmq\_bind(socket, endpoint) != 0) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_bind ");

exit(ERR\_ZMQ\_BIND);

}

}

void disconnect\_zmq\_socket(void\* socket, char\* endpoint) {

if (zmq\_disconnect(socket, endpoint) != 0) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_disconnect ");

exit(ERR\_ZMQ\_DISCONNECT);

}

}

void connect\_zmq\_socket(void\* socket, char\* endpoint) {

if (zmq\_connect(socket, endpoint) != 0) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_connect ");

exit(ERR\_ZMQ\_CONNECT);

}

}

void\* create\_zmq\_socket(void\* context, const int type) {

void\* socket = zmq\_socket(context, type);

if (socket == NULL) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_socket ");

exit(ERR\_ZMQ\_SOCKET);

}

return socket;

}

void reconnect\_zmq\_socket(void\* socket, int to, char\* addr) {

if (addr[16] != '\0') {

disconnect\_zmq\_socket(socket, addr);

}

create\_addr(addr, to);

connect\_zmq\_socket(socket, addr);

}

void close\_zmq\_socket(void\* socket) {

if (zmq\_close(socket) != 0) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_close ");

exit(ERR\_ZMQ\_CLOSE);

}

}

void destroy\_zmq\_context(void\* context) {

if (zmq\_ctx\_destroy(context) != 0) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_ctx\_destroy ");

exit(ERR\_ZMQ\_CLOSE);

}

}

int\* create\_vector() {

int\* ptr = malloc(sizeof(int));

if (ptr == NULL) return NULL;

ptr[0] = TERMINATOR;

return ptr;

}

int length(const int\* ptr) {

if (ptr == NULL) -1;

int i = 0;

while (ptr[i] != TERMINATOR) {

++i;

}

return i;

}

// very bad, but essy solution: not to use capacity

int\* push\_back(int\* ptr, int value) {

if (ptr == NULL) return NULL;

int size = 0;

while (ptr[size] != TERMINATOR) {

++size;

}

int\* tmp = realloc(ptr, (size + 2) \* sizeof(int));

if (tmp == NULL) {

free(ptr);

return NULL;

}

ptr = tmp;

ptr[size] = value;

ptr[size + 1] = TERMINATOR;

return ptr;

}

void erase(int\* ptr, int value) {

if (ptr == NULL) return;

int i = 0;

while (ptr[i] != TERMINATOR) {

if (ptr[i] == value) break;

++i;

}

// shift array after deleting the element

while (ptr[i] != TERMINATOR) {

ptr[i] = ptr[i + 1];

i++;

}

}

void destroy(int\* ptr) {

if (ptr == NULL) return;

free(ptr);

}

bool receive\_msg\_wait(void\* socket, message\* token) {

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

if (zmq\_msg\_recv(&reply, socket, 0) == -1) {

zmq\_msg\_close(&reply);

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_msg\_recv ");

exit(ERR\_ZMQ\_MSG);

}

(\*token) = \* ((message\*) zmq\_msg\_data(&reply));

if (zmq\_msg\_close(&reply) == -1) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_msg\_close ");

exit(ERR\_ZMQ\_MSG);

}

return true;

}

bool send\_msg\_wait(void\* socket, message\* token) {

zmq\_msg\_t msg;

zmq\_msg\_init(&msg);

if (zmq\_msg\_init\_size(&msg, sizeof(message)) == -1) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_msg\_init ");

exit(ERR\_ZMQ\_MSG);

}

if (zmq\_msg\_init\_data(&msg, token, sizeof(message), NULL, NULL) == -1) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_msg\_init ");

exit(ERR\_ZMQ\_MSG);

}

if (zmq\_msg\_send(&msg, socket, 0) == -1) {

zmq\_msg\_close(&msg);

return false;

}

return true;

}

bool ping\_process(int id) {

char addr\_monitor[30];

char addr\_connection[30];

char str[30];

memset(str, 0, 30);

sprintf(str, "%d", id);

memset(addr\_monitor, 0, 30);

memcpy(addr\_monitor, PING\_SOCKET\_PATTERN, sizeof(PING\_SOCKET\_PATTERN));

strcat(addr\_monitor, str);

memset(addr\_connection, 0, 30);

memcpy(addr\_connection, SERVER\_SOCKET\_PATTERN, sizeof(SERVER\_SOCKET\_PATTERN));

strcat(addr\_connection, str);

void\* context = zmq\_ctx\_new();

void \*requester = zmq\_socket(context, ZMQ\_REQ);

zmq\_socket\_monitor(requester, addr\_monitor, ZMQ\_EVENT\_CONNECTED | ZMQ\_EVENT\_CONNECT\_RETRIED);

void \*socet = zmq\_socket(context, ZMQ\_PAIR);

zmq\_connect(socet, addr\_monitor);

zmq\_connect(requester, addr\_connection);

zmq\_msg\_t msg;

zmq\_msg\_init(&msg);

zmq\_msg\_recv(&msg, socet, 0);

uint8\_t\* data = (uint8\_t\*)zmq\_msg\_data(&msg);

uint16\_t event = \*(uint16\_t\*)(data);

zmq\_close(requester);

zmq\_close(socet);

zmq\_msg\_close(&msg);

zmq\_ctx\_destroy(context);

if (event == ZMQ\_EVENT\_CONNECT\_RETRIED) {

return false;

} else {

return true;

}

}

=============================== msgQ.h ==================================

#ifndef LAB\_6\_MSGQ\_H

#define LAB\_6\_MSGQ\_H

#include <stdbool.h>

#include <string.h>

#include <unistd.h>

#include <stdlib.h>

#include <errno.h>

#include <zmq.h>

#include <assert.h>

#define MAX\_LEN 64

#define MN 30

#define ERR\_ZMQ\_CTX 100

#define ERR\_ZMQ\_SOCKET 101

#define ERR\_ZMQ\_BIND 102

#define ERR\_ZMQ\_CLOSE 102

#define ERR\_ZMQ\_CONNECT 104

#define ERR\_ZMQ\_DISCONNECT 105

#define ERR\_ZMQ\_MSG 106

#define SERVER\_SOCKET\_PATTERN "tcp://localhost:"

#define PING\_SOCKET\_PATTERN "inproc://ping"

#define TCP\_SOCKET\_PATTERN "tcp://\*:"

#define MIN\_ADDR 5555

#define TERMINATOR (-3000)

typedef enum {

create,

delete,

exec,

success

} cmd\_type;

typedef struct {

cmd\_type cmd;

int value;

char str[MAX\_LEN];

char sub[MAX\_LEN];

int res[MAX\_LEN];

} message;

void clear\_token(message\* msg);

void create\_addr(char\* addr, int id);

void bind\_zmq\_socket(void\* socket, char\* endpoint);

void\* create\_zmq\_context();

void\* create\_zmq\_socket(void\* context, const int type);

void connect\_zmq\_socket(void\* socket, char\* endpoint);

void disconnect\_zmq\_socket(void\* socket, char\* endpoint);

void reconnect\_zmq\_socket(void\* socket, int to, char\* addr);

void close\_zmq\_socket(void\* socket);

void destroy\_zmq\_context(void\* context);

bool receive\_msg\_wait(void\* socket, message\* token);

bool send\_msg\_wait(void\* socket, message\* token);

bool ping\_process(int id);

// TODO move implementation "std:vector<int>" to a special .h and .c files

int\* create\_vector();

int length(const int\* ptr);

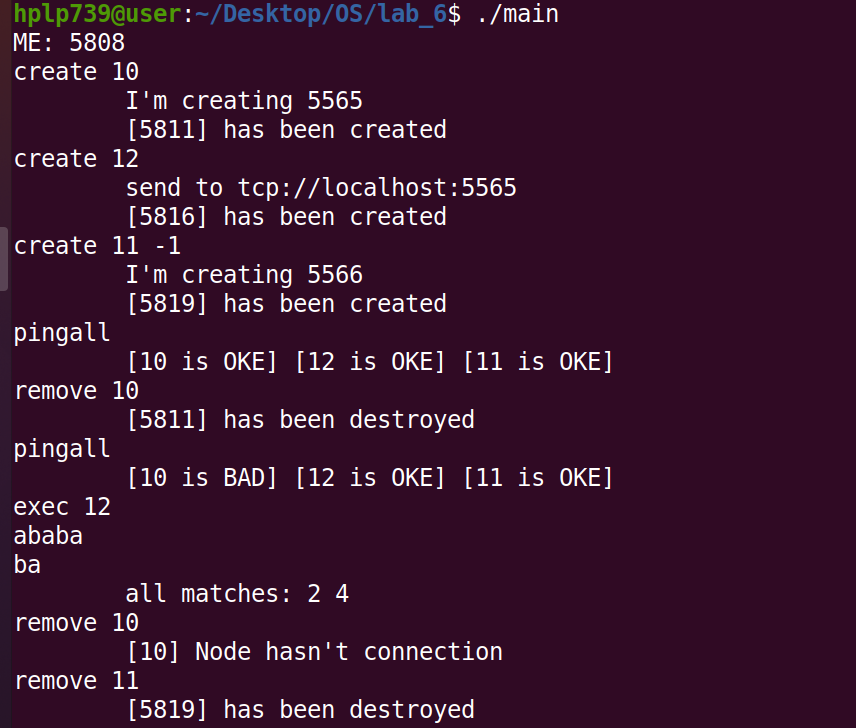
int\* push\_back(int\* ptr, int value);

void erase(int\* ptr, int value);

void destroy(int\* ptr);

#endif //LAB\_6\_MSGQ\_H

**Демонстрация работы программы**



**Выводы**

В ходе выполнения лабораторной работы изучил основы работы с очередями сообщений ZMQ и реализовал программу с использованием этой библиотеки. В процессе открыл для себя много нового. Почти вся информация бралась из официальной документации.